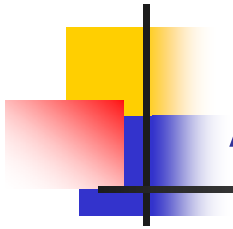
TSW – Reliability and Fault Tolerance

Alexandre David

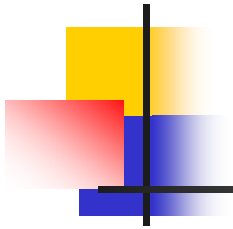
1.2.05

Credits: some slides by Alan Burns & Andy Wellings.



Aims

- Understand the factors which affect the reliability of a system.
- Introduce how software design faults can be tolerated.
- Concepts:
  - Safety and Dependability
  - Reliability, failure and faults
  - Failure modes
  - Fault prevention and fault tolerance
  - N-Version programming
  - Dynamic Redundancy



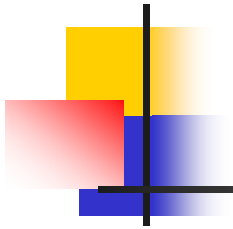
Sources of faults

- Inadequate specifications.
- Design errors in software.
- Hardware failure.
- Interference on the communication sub-system.

Safety and reliability ✓

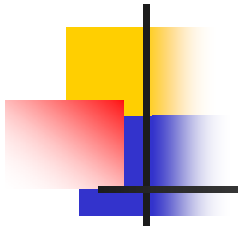
- **Safety**: freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm
  - Most systems that have an element of risk associated with their use are unsafe.
- **Reliability**: a measure of how well a system conforms to the specification of its behavior.

- Safety is the probability that conditions that can lead to mishaps do not occur whether or not the intended function is performed.



Safety and reliability

- A plane that never flies is very safe but unreliable.
- Nuclear bombs are very reliable demolition devices but very unsafe.
- Increasing the likelihood to fire a weapon increases reliability but decreases safety.

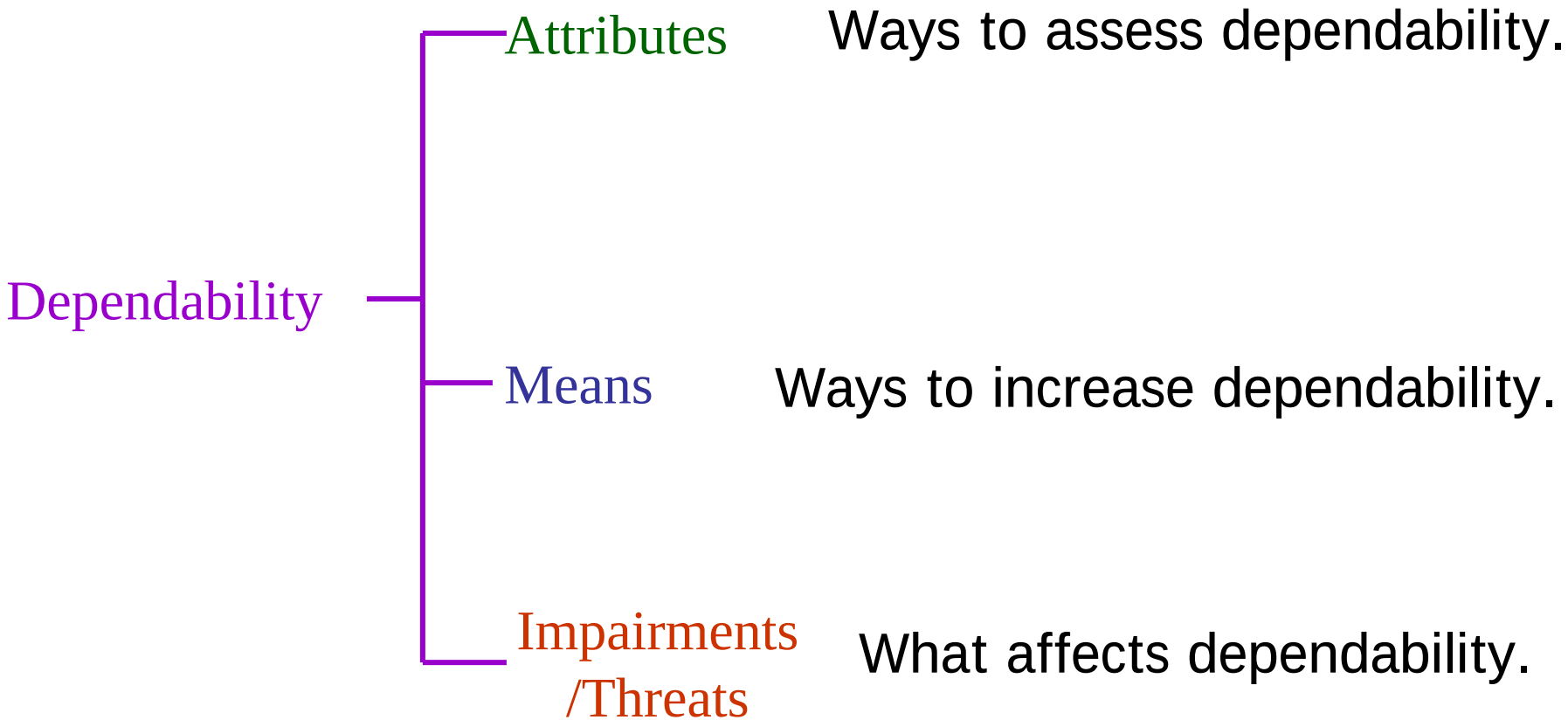


Dependability

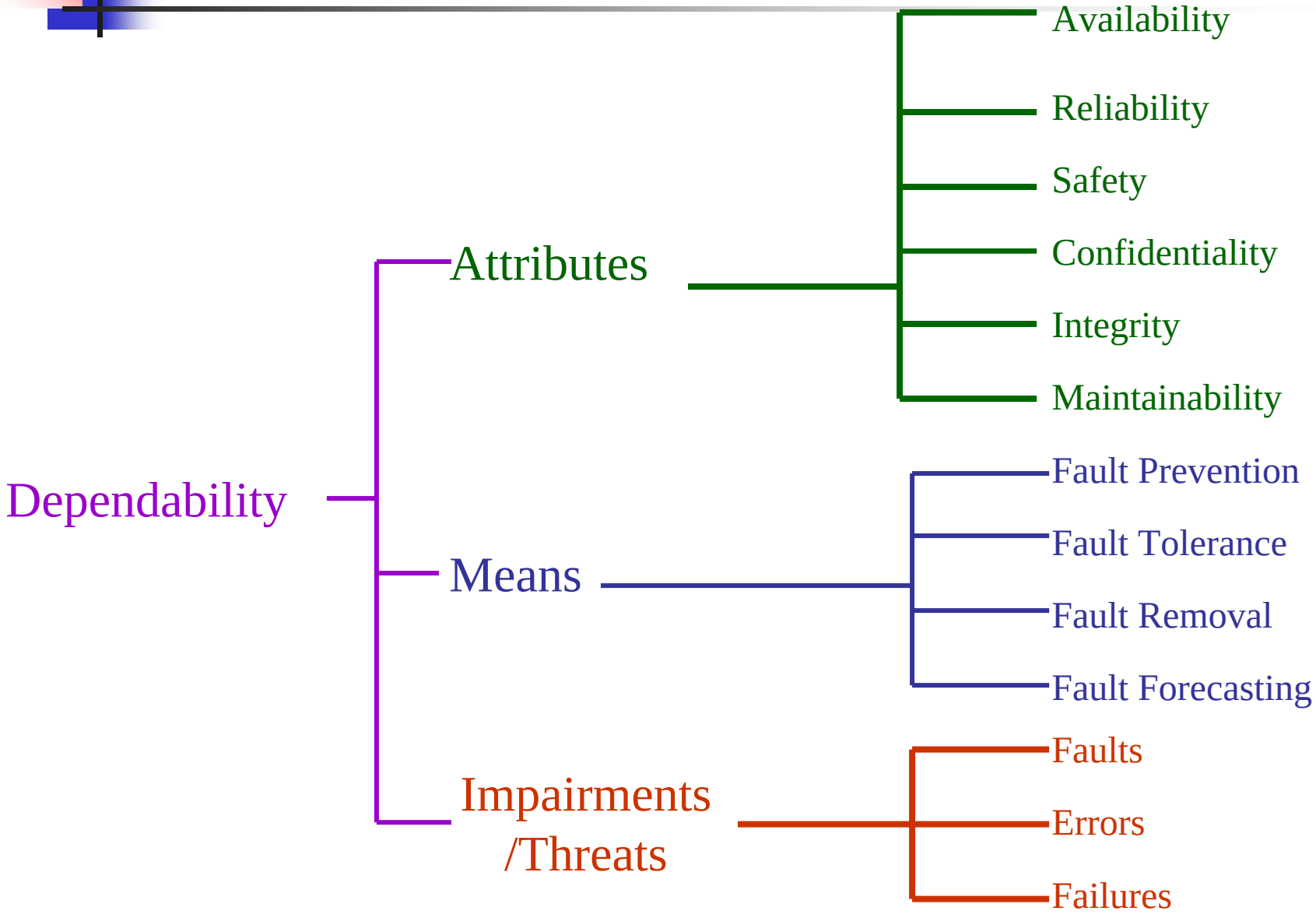
- *Dependability* as applied to a computer system is defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as:
  - "*[..] the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [..]*"

- General notion that encompasses security, reliability, safety, fault tolerance...

Dependability terminology



Dependability terminology

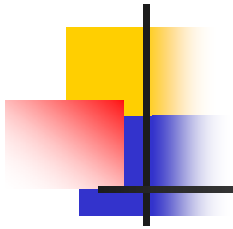


Reliability, failure, and faults

- Reliability: how well a system conforms to its specified behavior.
  - Deviation = failure.
    - Failures are caused by all sorts of problems and show themselves by unexpected external behaviors.
  - The problems are called errors.
  - Their causes are called faults.

Types of faults

- Transient faults: they appear at some point, stay, and they disappear. They are caused by temporary external events.
  - e.g. communication while crossing a tunnel...
- Permanent faults: they remain in the system until repaired.
  - e.g. broken cables.
- Intermittent faults: they occur from time to time. They are caused by recurring events.
  - e.g. overheating component.

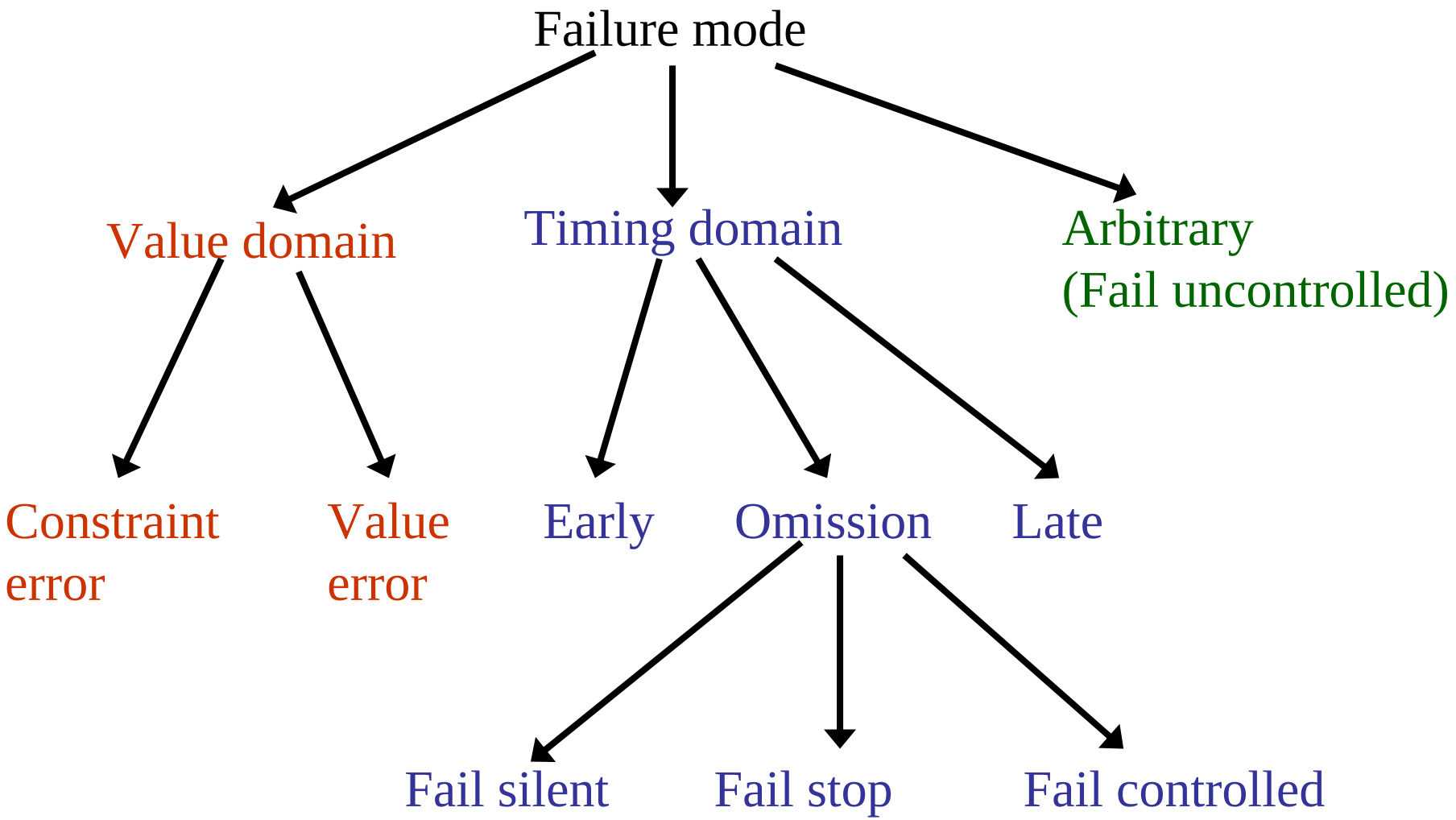


Software faults

- A.k.a. bugs.
  - Bohrbugs: reproducible and identified.
  - Heisenbugs: occur in rare conditions, usually disappear upon inspection.

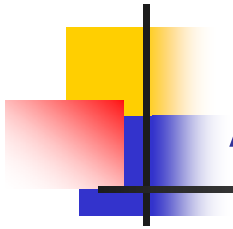
- Discuss: *Software does not deteriorate with age, it is either correct or incorrect*.
  - Faults can remain dormant.
  - Software is reused.

Failure modes



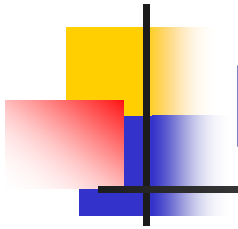
Byzantine failure

- Byzantine fault is an arbitrary fault that causes omission failure or commission failure (incorrect answer, corrupt data).
  - Byzantine failure models a network of processes where some of them fail. The problem is to detect which ones.
- Byzantine Generals' Problem has no solution unless $n > 3t$, where n is the number of processes in the system and t the number of faulty processes (a.k.a. resilience of the algorithm).



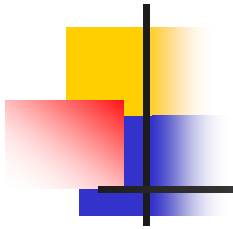
Achieving reliable systems

- Fault prevention attempts to eliminate any possibility of faults creeping into a system before it goes operational.
- Fault tolerance enables a system to continue functioning even in the presence of faults.
- Both approaches: define failure modes.



Fault prevention

- Fault avoidance and fault removal.
- Hints for fault avoidance:
  - use most reliable components (@ fixed cost)
  - use refined techniques to assemble sub-systems, hierarchy
  - package & shield hardware from interferences
  - use rigorous/formal specification of requirements
  - use proven design methodologies
  - choose language offering data abstraction and modularity (e.g. encapsulation)
  - use tools and environment to manage complexity

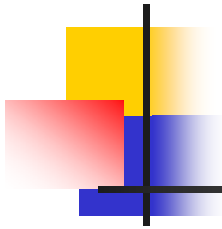


Fault removal

- Find faults and then remove them
  - use program verification, code inspection, testing
- Testing: not exhaustive, only gives confidence with some probabilities.
  - can find faults but not prove their absence
  - testing may be impossible, only simulation
  - problem of accuracy: simulation $\leftrightarrow$ reality
  - requirement errors may be discovered once the system is operational

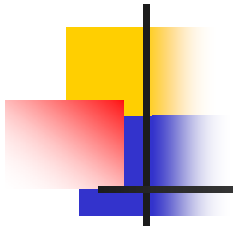
Fault tolerance

- Faults may still occur, uncontrollable or unavoidable.
- Levels of fault tolerance:
  - Full fault tolerance: the system continues to work unaffected for a limited period of time.
  - Graceful degradation: the system continues to work with degraded performance or functionality.
  - Fail safe: the system stops to work but returns to a safe state before to maintain its integrity.



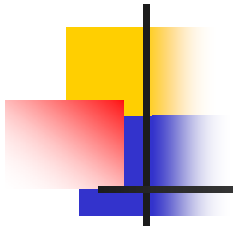
Redundancy

- Fault tolerant techniques rely on redundancy: duplicated hardware.
  - The catch: to detect and recover, you need more hardware & software → more complexity → less reliability → more faults.
  - Goal: minimize redundancy and maximize reliability.



Hardware fault tolerance

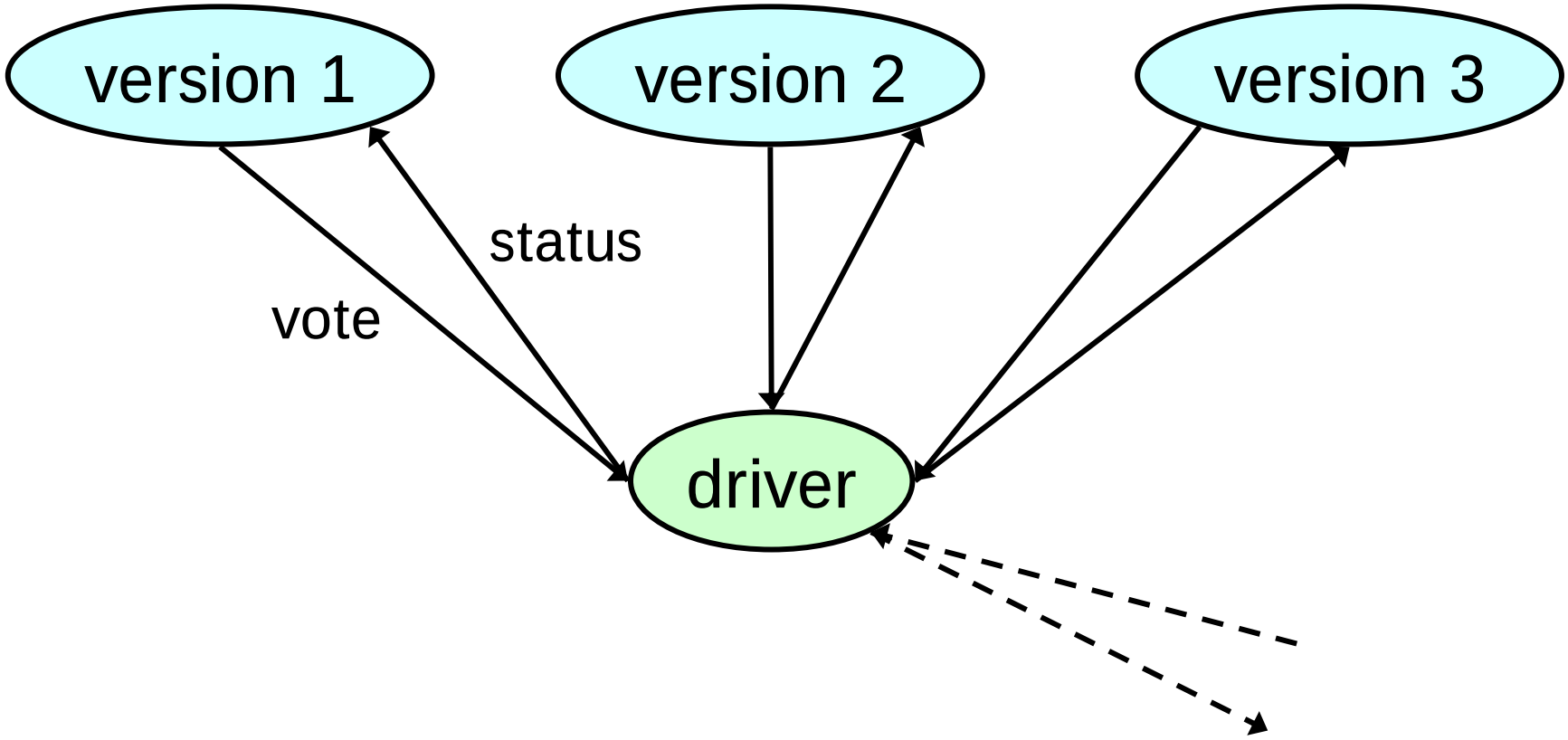
- Static: redundant components.
  - If one has a fault, the others mask it.
  - Voting techniques.
  - Triple/N modular redundancy.
- Dynamic: redundancy inside a component.
  - checksums, parity bits...

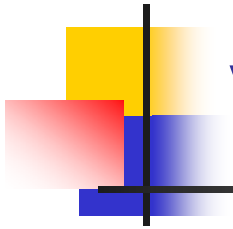


Software fault tolerance

- Static: N-version programming
  - Design diversity – counter design errors.
    - **Independent** developments.
    - Better if different languages etc...
    - Cost problem.
  - Programs execute concurrently and the final result is voted.
  - Assume: Programs developed independently will fail independently.
- Dynamic: detect and recover.
  - Detection: HW, OS, replication checks, asserts, redundancy...
  - Backward error recovery (unroll) with recovery blocks.
  - Forward error recovery (correct) with exceptions.

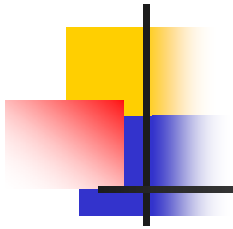
N-version programming





Votes

- Problems with comparisons
  - text or integers easy
  - floating point numbers tricky
  - $\rightarrow$ inexact voting techniques
  - Even so: consistency problems with threshold
    - A chooses to open a valve
    - B chooses to close it
    - values are close but decisions are conflicting

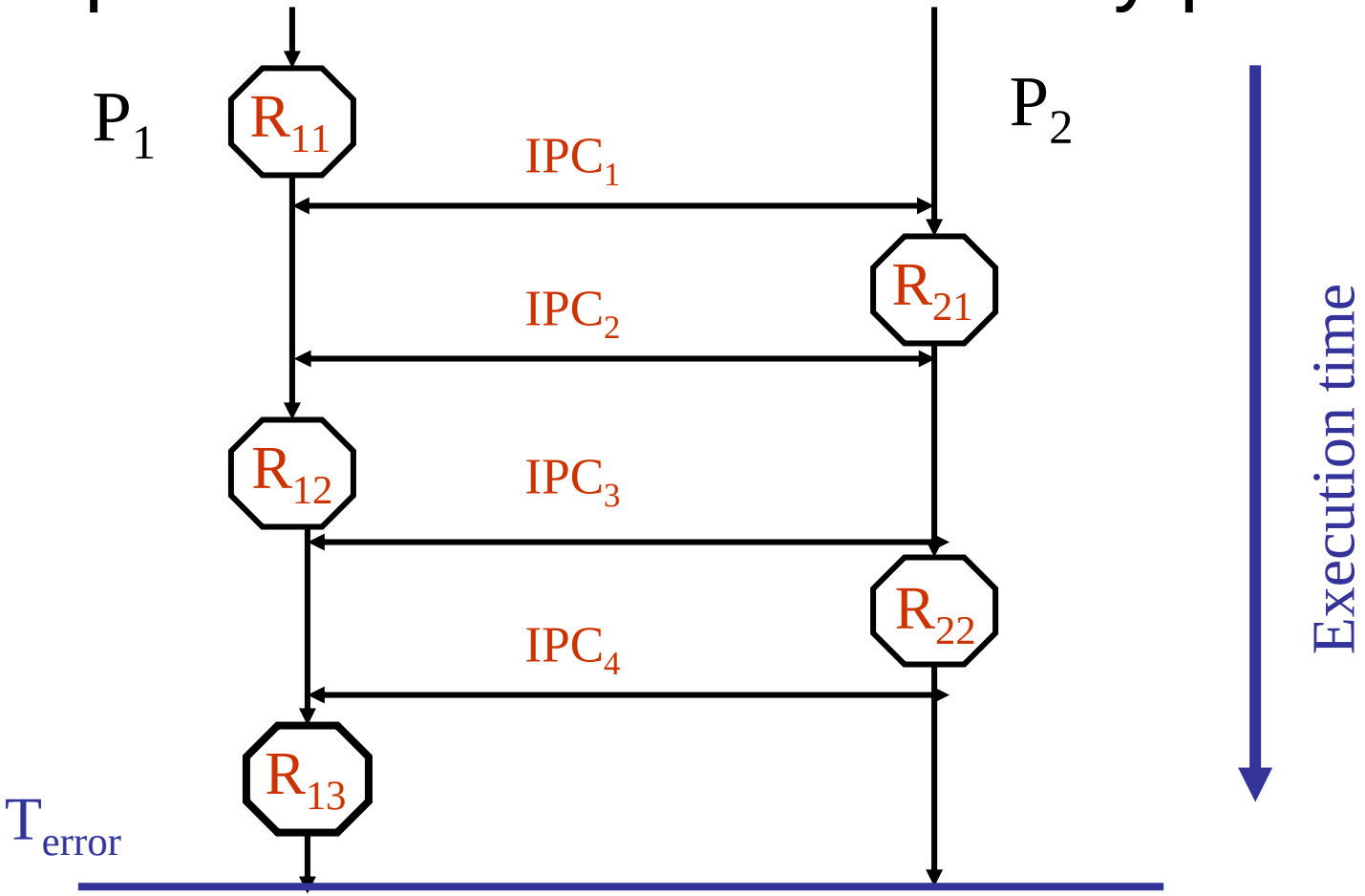


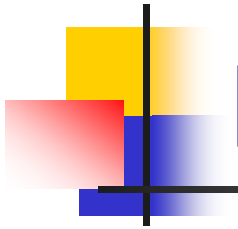
Error recovery

- Go back or go to a consistent state
  - allows to *confine* the damage
  - easier with modular decomposition
- Forward recovery: try {.. } catch(){ correct }
- Backward recovery:
  - go back to a *recovery point* by *checkpointing*,
  - but can’t undo everything, e.g., a lightning bolt that fries a component,
  - domino effect

Domino effect

- Concurrent processes interact
  → dependencies between recovery points.

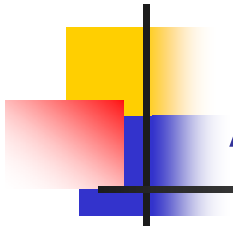




Recovery blocks

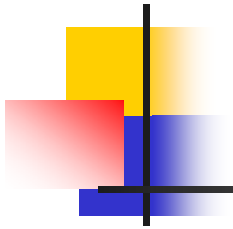
- Entrance: recovery point.
- Exit: acceptance test.
  - If fails, recover to recovery point and execute an alternative module (don't make the same mistake again!).
  - If all alternatives are exhausted, propagate the error at a higher level.
  - E.g. try different techniques to solve an equation. Fast 1st but imprecise, if too bad try more expensive.

```
ensure <acceptance test>
by
    <primary module>
else by
    <alternative module>
    ...
else by
    <alternative module>
else error
```



Acceptance test

- Provides the error detection mechanism.
  - Problem: Limit its overhead.
  - Faulty/imprecise test may leave residual errors.
- Acceptance test – not correctness.
  - Allow components to offer degraded services.



N-version programming vs. recovery blocks

- Static vs. dynamic.
- Overhead: N*resources vs. recovery points.
  - Error detection: voting vs. acceptance test.
- Atomicity: NV votes and then outputs, RB outputs after passing the test.
- Common: alternative algorithms, both vulnerable to errors in requirements.

Concept summary

- Dependability, safety, reliability, failure, faults.
- Fault prevention consists of fault avoidance and fault removal.
- Fault tolerance: static and dynamic
- N-version programming: the independent generation of N functionally equivalent programs from the same specification.
  - Assume: Programs developed independently will fail independently.
- Dynamic redundancy: detect & recover, forward/backward error recovery.